**UNIT-V**

Graph

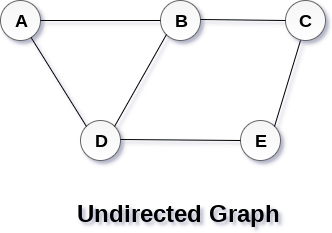
A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Definition

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

vertices and E(G) represents the set of edges which are used to connect these vertices.

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.

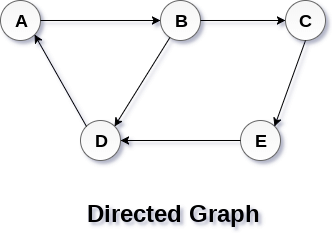


Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure.



**Graph Terminology**

Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if V0=VN.

Simple Path

If all the nodes of the graph are distinct with an exception V0=VN, then such path P is called as closed simple path.

Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop

An edge that is associated with the similar end points can be called as Loop.

Adjacent Nodes

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

**Graph representation**

A graph is a data structure that consist a sets of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:

Sequential representation (or, Adjacency matrix representation)

Linked list representation (or, Adjacency list representation)

In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked list representation, there is a use of an adjacency list to store the graph.

In this tutorial, we will discuss each one of them in detail.

Now, let's start discussing the ways of representing a graph in the data structure.

Sequential representation

In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

An entry Aij in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between Vi and Vj. If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is n x n, and the matrix A = [aij] can be defined as -

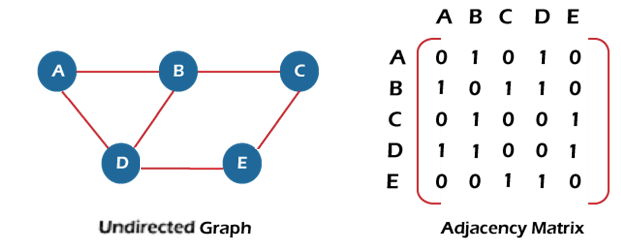
aij = 1 {if there is a path exists from Vi to Vj}

aij = 0 {Otherwise}

It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes, whereas 1 represents the existence of a path between two edges.

If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be 0.

Now, let's see the adjacency matrix representation of an undirected graph.



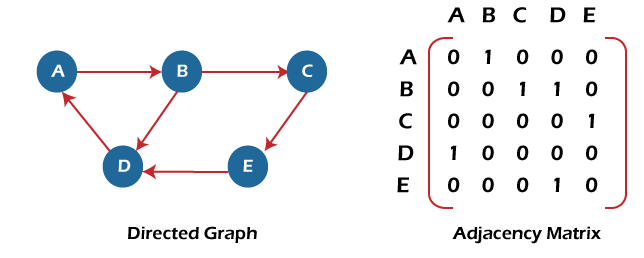
In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix.

There exist different adjacency matrices for the directed and undirected graph. In a directed graph, an entry Aij will be 1 only when there is an edge directed from Vi to Vj.

**Adjacency matrix for a directed graph**

In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.

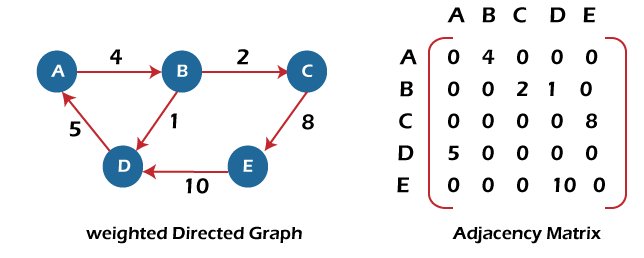
Consider the below-directed graph and try to construct the adjacency matrix of it.



In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.

Adjacency matrix for a weighted directed graph

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix. We can understand it with the help of an example. Consider the below graph and its adjacency matrix representation. In the representation, we can see that the weight associated with the edges is represented as the entries in the adjacency matrix.



In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

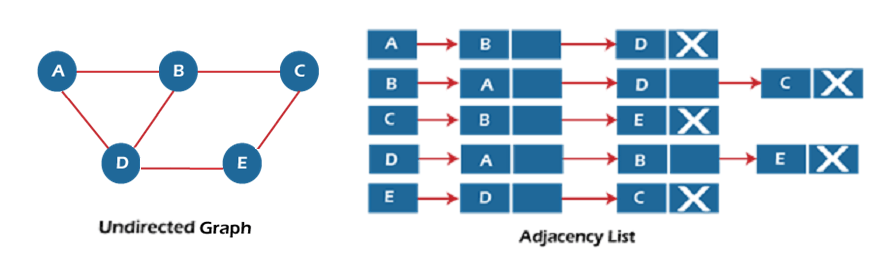
Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large.

Though, it is advantageous to use an adjacency matrix, but it consumes more space. Even if the graph is sparse, the matrix still consumes the same space.

**Linked list representation**

An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.

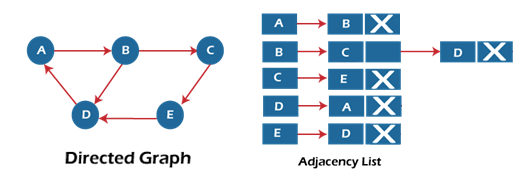


In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

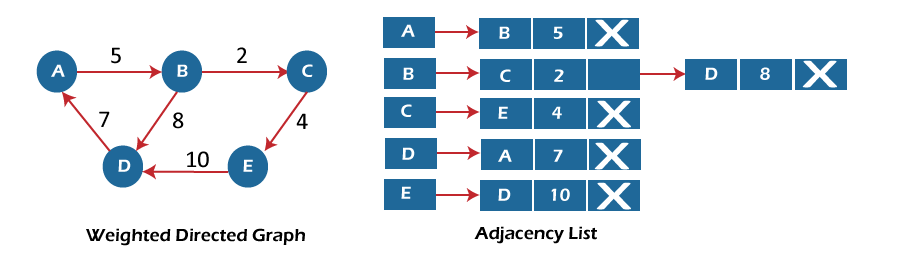
The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

Now, consider the directed graph, and let's see the adjacency list representation of that graph.



For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.



In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.

In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space.

Implementation of adjacency matrix representation of Graph

Now, let's see the implementation of adjacency matrix representation of graph in C.

In this program, there is an adjacency matrix representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

Here, there are four vertices and five edges in the graph that are non-directed.

/\* Adjacency Matrix representation of an undirected graph in C \*/

  #include <stdio.h>

#define V 4 /\* number of vertices in the graph \*/

  /\* function to initialize the matrix to zero \*/

void init(int arr[][V]) {

  int i, j;

  for (i = 0; i < V; i++)

    for (j = 0; j < V; j++)

      arr[i][j] = 0;

}

  /\* function to add edges to the graph \*/

void insertEdge(int arr[][V], int i, int j) {

  arr[i][j] = 1;

  arr[j][i] = 1;

}

  /\* function to print the matrix elements \*/

void printAdjMatrix(int arr[][V]) {

  int i, j;

  for (i = 0; i < V; i++) {

    printf("%d: ", i);

    for (j = 0; j < V; j++) {

      printf("%d ", arr[i][j]);

    }

    printf("\n");

  }

}

int main() {

  int adjMatrix[V][V];

  init(adjMatrix);

  insertEdge(adjMatrix, 0, 1);

  insertEdge(adjMatrix, 0, 2);

  insertEdge(adjMatrix, 1, 2);

  insertEdge(adjMatrix, 2, 0);

  insertEdge(adjMatrix, 2, 3);

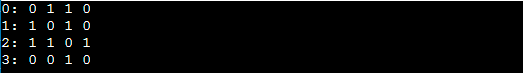
   printAdjMatrix(adjMatrix);

    return 0;

}

Output:

After the execution of the above code, the output will be -



**Degree of a vertex**

**In-degree of a vertex**

The in-degree of a vertex can be described as a number of edges with v, where v is used to indicate the terminal vertex. In other words, we can describe it as a number of edges coming to the vertex. With the help of syntax deg-(v), we can write the in-degree of a vertex. If we want to determine the in-degree of a vertex, for this, we have to count the number of edges that ends at the vertex.

**Out-degree of a vertex**

The out-degree of a vertex can be described as a number of edges with v, where v is used to indicate the initial vertex. In other words, we can describe it as a number of edges coming out from the vertex. With the help of syntax deg+(v), we can write the out-degree of a vertex. If we want to determine the out-degree of a vertex, for this, we have to count the number of edges that begins from the vertex.

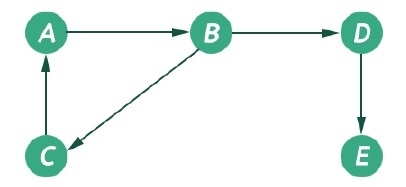
## Applications of Graphs in Data Structure

Graphs data structures have a variety of applications. Some of the most popular applications are:

* Helps to define the flow of computation of software programs.
* Used in Google maps for building transportation systems. In google maps, the intersection of two or more roads represents the node while the road connecting two nodes represents an edge. Google maps algorithm uses graphs to calculate the shortest distance between two vertices.
* Used in social networks such as Facebook and Linkedin.
* [Operating Systems](https://www.naukri.com/learning/what-is-operating-system-st617) use Resource Allocation Graph where every process and resource acts as a node while edges are drawn from resources to the allocated process.
* Used in the world wide web where the web pages represent the nodes.
* [Blockchains](https://www.naukri.com/learning/what-is-blockchain-st561) also use graphs. The nodes are blocks that store many transactions while the edges connect subsequent blocks.
* Used in modeling data.

**Traversal in Graph**

The graph is one non-linear data structure. That is consists of some nodes and their connected edges. The edges may be director or undirected. This graph can be represented as G(V, E). The following graph can be represented as G({A, B, C, D, E}, {(A, B), (B, D), (D, E), (B, C), (C, A)})



The graph has two types of traversal algorithms. These are called the Breadth First Search and Depth First Search.

**Breadth First Search (BFS)**

The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph. In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one. After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again.

Algorithm

bfs(vertices, start)

Input: The list of vertices, and the start vertex.

Output: Traverse all of the nodes, if the graph is connected.

Begin

   define an empty queue que

   at first mark all nodes status as unvisited

   add the start vertex into the que

   while que is not empty, do

      delete item from que and set to u

      display the vertex u

      for all vertices 1 adjacent with u, do

         if vertices[i] is unvisited, then

            mark vertices[i] as temporarily visited

            add v into the queue

         mark

      done

      mark u as completely visited

   done

End

**Depth First Search (DFS)**

The Depth First Search (DFS) is a graph traversal algorithm. In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner.

Algorithm

dfs(vertices, start)

Input: The list of all vertices, and the start node.

Output: Traverse all nodes in the graph.

Begin

   initially make the state to unvisited for all nodes

   push start into the stack

   while stack is not empty, do

      pop element from stack and set to u

      display the node u

      if u is not visited, then

         mark u as visited

         for all nodes i connected to u, do

            if ith vertex is unvisited, then

               push ith vertex into the stack

               mark ith vertex as visited

         done

   done

End.

**Compare the BFS and DFS:**

| **Sr. No.** | **Key** | **BFS** | **DFS** |
| --- | --- | --- | --- |
| 1 | Definition | BFS, stands for Breadth First Search. | DFS, stands for Depth First Search. |
| 2 | Data structure | BFS uses Queue to find the shortest path. | DFS uses Stack to find the shortest path. |
| 3 | Source | BFS is better when target is closer to Source. | DFS is better when target is far from source. |
| 4 | Suitablity for decision tree | As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games. | DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won. |
| 5 | Speed | BFS is slower than DFS. | DFS is faster than BFS. |
| 6 | Time Complexity | Time Complexity of BFS = O(V+E) where V is vertices and E is edges. | Time Complexity of DFS is also O(V+E) where V is vertices and E is edges. |

**What is a spanning tree?**

A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.

A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.

A complete undirected graph can have nn-2 number of spanning trees where n is the number of vertices in the graph. Suppose, if n = 5, the number of maximum possible spanning trees would be 55-2 = 125.

**Applications of the spanning tree**

Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows -

Cluster Analysis

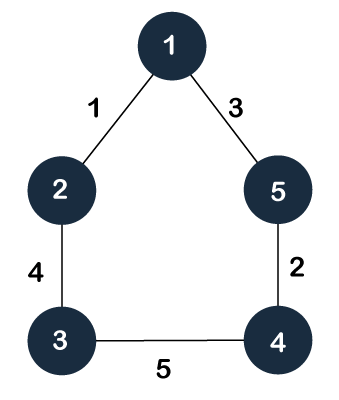
Civil network planning

Computer network routing protocol

Now, let's understand the spanning tree with the help of an example.

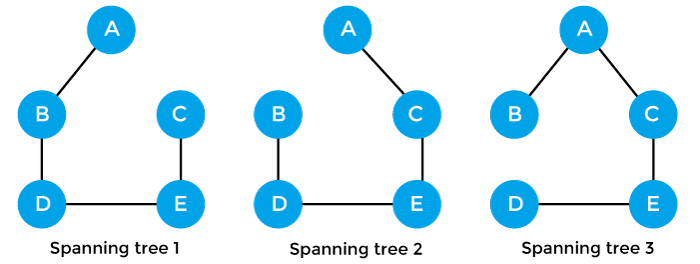
Example of Spanning tree

Suppose the graph be -



As discussed above, a spanning tree contains the same number of vertices as the graph, the number of vertices in the above graph is 5; therefore, the spanning tree will contain 5 vertices. The edges in the spanning tree will be equal to the number of vertices in the graph minus 1. So, there will be 4 edges in the spanning tree.

Some of the possible spanning trees that will be created from the above graph are given as follows -



Properties of spanning-tree

Some of the properties of the spanning tree are given as follows -

There can be more than one spanning tree of a connected graph G.

A spanning tree does not have any cycles or loop.

A spanning tree is minimally connected, so removing one edge from the tree will make the graph disconnected.

A spanning tree is maximally acyclic, so adding one edge to the tree will create a loop.

There can be a maximum nn-2 number of spanning trees that can be created from a complete graph.

A spanning tree has n-1 edges, where 'n' is the number of nodes.

If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.

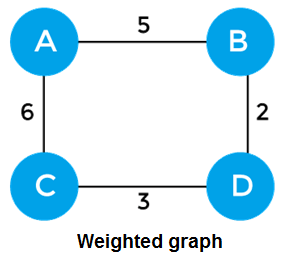
So, a spanning tree is a subset of connected graph G, and there is no spanning tree of a disconnected graph.

**Minimum Spanning tree**

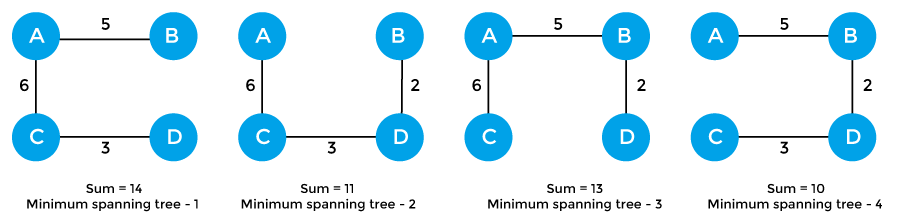
A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree. In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.

Example of minimum spanning tree

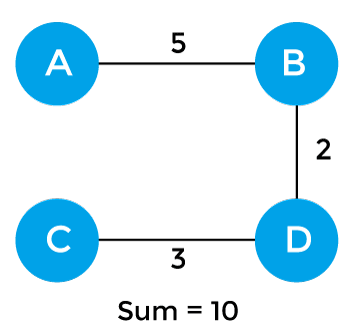
Let's understand the minimum spanning tree with the help of an example.



The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are -



So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is -



Applications of minimum spanning tree

The applications of the minimum spanning tree are given as follows -

Minimum spanning tree can be used to design water-supply networks, telecommunication networks, and electrical grids.

It can be used to find paths in the map.

Algorithms for Minimum spanning tree

A minimum spanning tree can be found from a weighted graph by using the algorithms given below -

**Prim's Algorithm& Kruskal's Algorithm**

**Prim's algorithm** - It is a greedy algorithm that starts with an empty spanning tree. It is used to find the minimum spanning tree from the graph. This algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

**Kruskal's algorithm** - This algorithm is also used to find the minimum spanning tree for a connected weighted graph. Kruskal's algorithm also follows greedy approach, which finds an optimum solution at every stage instead of focusing on a global optimum.

**Prim's Algorithm**

Before starting the main topic, we should discuss the basic and important terms such as spanning tree and minimum spanning tree.

Spanning tree - A spanning tree is the subgraph of an undirected connected graph.

Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Now, let's start the main topic.

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

How does the prim's algorithm work?

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

First, we have to initialize an MST with the randomly chosen vertex.

Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.

Repeat step 2 until the minimum spanning tree is formed.

The applications of prim's algorithm are -

Prim's algorithm can be used in network designing.

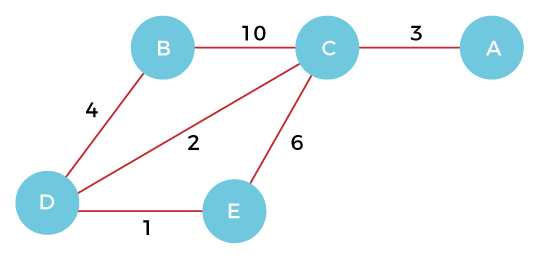
It can be used to make network cycles.

It can also be used to lay down electrical wiring cables.

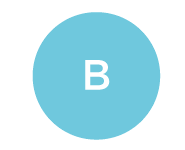
Example of prim's algorithm

Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

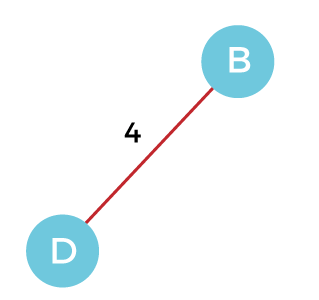
Suppose, a weighted graph is -



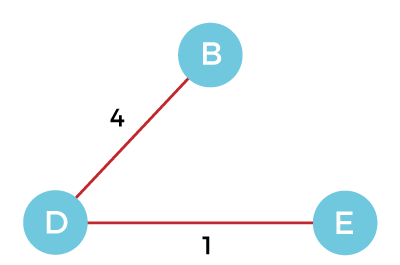
Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.



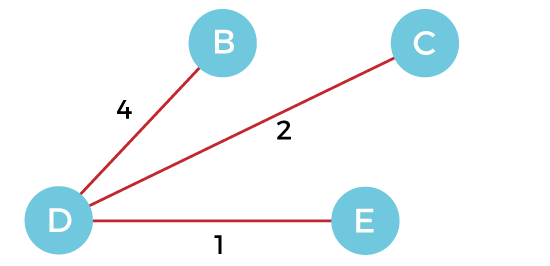
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



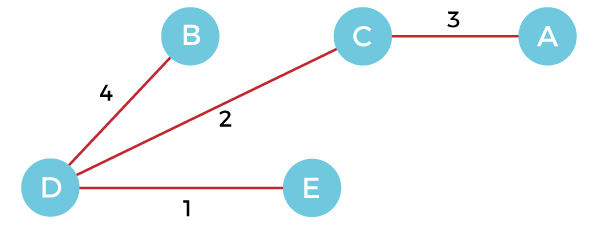
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

**Algorithm**

Step 1: Select a starting vertex

Step 2: Repeat Steps 3 and 4 until there are fringe vertices

Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight

Step 4: Add the selected edge and the vertex to the minimum spanning tree T

[END OF LOOP]

Step 5: EXIT

**Kruskal's Algorithm**

In this article, we will discuss Kruskal's algorithm. Here, we will also see the complexity, working, example, and implementation of the Kruskal's algorithm.

But before moving directly towards the algorithm, we should first understand the basic terms such as spanning tree and minimum spanning tree.

Spanning tree - A spanning tree is the subgraph of an undirected connected graph.

Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Now, let's start with the main topic.

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

How does Kruskal's algorithm work?

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

First, sort all the edges from low weight to high.

Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.

Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

The applications of Kruskal's algorithm are -

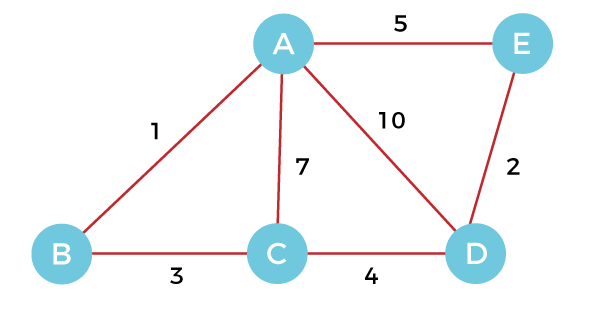
Kruskal's algorithm can be used to layout electrical wiring among cities.

It can be used to lay down LAN connections.

Example of Kruskal's algorithm

Now, let's see the working of Kruskal's algorithm using an example. It will be easier to understand Kruskal's algorithm using an example.

Suppose a weighted graph is -



The weight of the edges of the above graph is given in the below table -

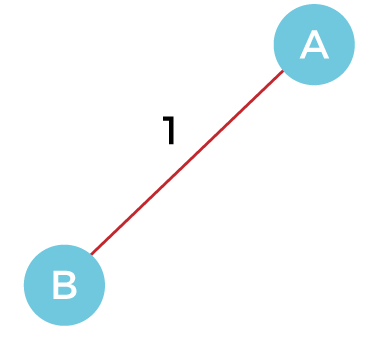
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Edge | AB | AC | AD | AE | BC | CD | DE |
| Weight | 1 | 7 | 10 | 5 | 3 | 4 | 2 |

Now, sort the edges given above in the ascending order of their weights.

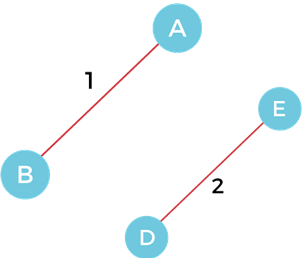
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Edge | AB | DE | BC | CD | AE | AC | AD |
| Weight | 1 | 2 | 3 | 4 | 5 | 7 | 10 |

Now, let's start constructing the minimum spanning tree.

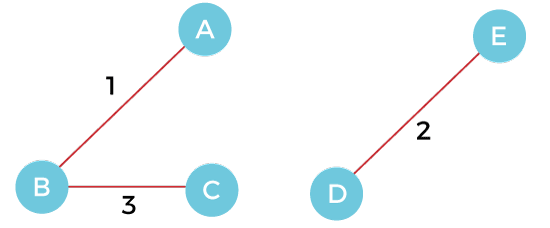
Step 1 - First, add the edge AB with weight 1 to the MST.



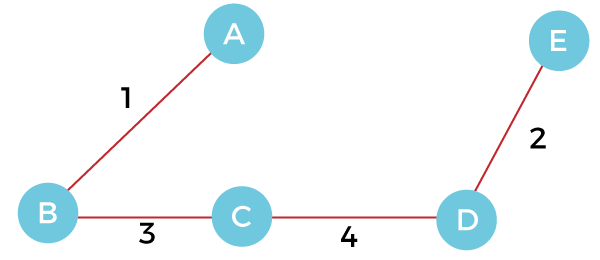
Step 2 - Add the edge DE with weight 2 to the MST as it is not creating the cycle.



Step 3 - Add the edge BC with weight 3 to the MST, as it is not creating any cycle or loop.



Step 4 - Now, pick the edge CD with weight 4 to the MST, as it is not forming the cycle.

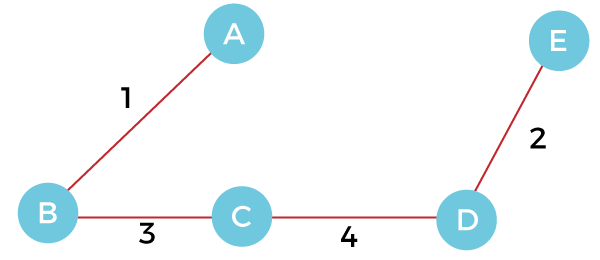


Step 5 - After that, pick the edge AE with weight 5. Including this edge will create the cycle, so discard it.

Step 6 - Pick the edge AC with weight 7. Including this edge will create the cycle, so discard it.

Step 7 - Pick the edge AD with weight 10. Including this edge will also create the cycle, so discard it.

So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is -



The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.

Now, the number of edges in the above tree equals the number of vertices minus 1. So, the algorithm stops here.

Algorithm

Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.

Step 2: Create a set E that contains all the edges of the graph.

Step 3: Repeat Steps 4 and 5 while E is NOT EMPTY and F is not spanning

Step 4: Remove an edge from E with minimum weight

Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F

(for combining two trees into one tree).

ELSE

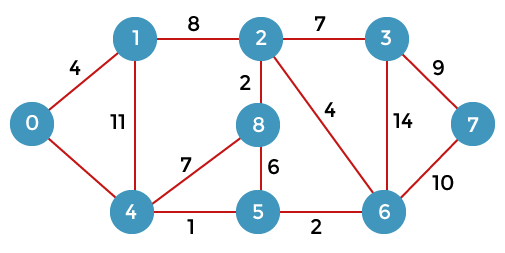
Discard the edge

Step 6: END

**Dijkstra Algorithm**

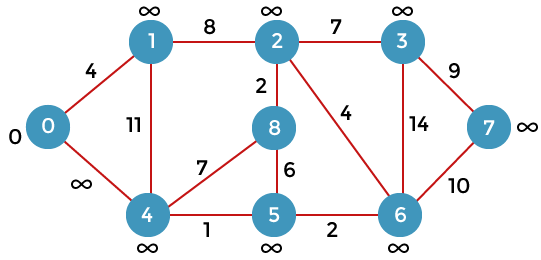
Dijkstra algorithm is a single-source shortest path algorithm. Here, single-source means that only one source is given, and we have to find the shortest path from the source to all the nodes.

Let's understand the working of Dijkstra's algorithm. Consider the below graph.



First, we have to consider any vertex as a source vertex. Suppose we consider vertex 0 as a source vertex.

Here we assume that 0 as a source vertex, and distance to all the other vertices is infinity. Initially, we do not know the distances. First, we will find out the vertices which are directly connected to the vertex 0. As we can observe in the above graph that two vertices are directly connected to vertex 0.



Let's assume that the vertex 0 is represented by 'x' and the vertex 1 is represented by 'y'. The distance between the vertices can be calculated by using the below formula:

d(x, y) = d(x) + c(x, y) < d(y)

= (0 + 4) < ∞

= 4 < ∞

Since 4<∞ so we will update d(v) from ∞ to 4.

Therefore, we come to the conclusion that the formula for calculating the distance between the vertices:

{if( d(u) + c(u, v) < d(v))

d(v) = d(u) +c(u, v) }

Now we consider vertex 0 same as 'x' and vertex 4 as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (0 + 8) < ∞

= 8 < ∞

Therefore, the value of d(y) is 8. We replace the infinity value of vertices 1 and 4 with the values 4 and 8 respectively. Now, we have found the shortest path from the vertex 0 to 1 and 0 to 4. Therefore, vertex 0 is selected. Now, we will compare all the vertices except the vertex 0. Since vertex 1 has the lowest value, i.e., 4; therefore, vertex 1 is selected.

Since vertex 1 is selected, so we consider the path from 1 to 2, and 1 to 4. We will not consider the path from 1 to 0 as the vertex 0 is already selected.

First, we calculate the distance between the vertex 1 and 2. Consider the vertex 1 as 'x', and the vertex 2 as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (4 + 8) < ∞

= 12 < ∞

Since 12<∞ so we will update d(2) from ∞ to 12.

Now, we calculate the distance between the vertex 1 and vertex 4. Consider the vertex 1 as 'x' and the vertex 4 as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (4 + 11) < 8

= 15 < 8

Since 15 is not less than 8, we will not update the value d(4) from 8 to 12.

Till now, two nodes have been selected, i.e., 0 and 1. Now we have to compare the nodes except the node 0 and 1. The node 4 has the minimum distance, i.e., 8. Therefore, vertex 4 is selected.

Since vertex 4 is selected, so we will consider all the direct paths from the vertex 4. The direct paths from vertex 4 are 4 to 0, 4 to 1, 4 to 8, and 4 to 5. Since the vertices 0 and 1 have already been selected so we will not consider the vertices 0 and 1. We will consider only two vertices, i.e., 8 and 5.

First, we consider the vertex 8. First, we calculate the distance between the vertex 4 and 8. Consider the vertex 4 as 'x', and the vertex 8 as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (8 + 7) < ∞

= 15 < ∞

Since 15 is less than the infinity so we update d(8) from infinity to 15.

Now, we consider the vertex 5. First, we calculate the distance between the vertex 4 and 5. Consider the vertex 4 as 'x', and the vertex 5 as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (8 + 1) < ∞

= 9 < ∞

Since 5 is less than the infinity, we update d(5) from infinity to 9.

Till now, three nodes have been selected, i.e., 0, 1, and 4. Now we have to compare the nodes except the nodes 0, 1 and 4. The node 5 has the minimum value, i.e., 9. Therefore, vertex 5 is selected.

Since the vertex 5 is selected, so we will consider all the direct paths from vertex 5. The direct paths from vertex 5 are 5 to 8, and 5 to 6.

First, we consider the vertex 8. First, we calculate the distance between the vertex 5 and 8. Consider the vertex 5 as 'x', and the vertex 8 as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (9 + 15) < 15

= 24 < 15

Since 24 is not less than 15 so we will not update the value d(8) from 15 to 24.

Now, we consider the vertex 6. First, we calculate the distance between the vertex 5 and 6. Consider the vertex 5 as 'x', and the vertex 6 as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (9 + 2) < ∞

= 11 < ∞

Since 11 is less than infinity, we update d(6) from infinity to 11.

Till now, nodes 0, 1, 4 and 5 have been selected. We will compare the nodes except the selected nodes. The node 6 has the lowest value as compared to other nodes. Therefore, vertex 6 is selected.

Since vertex 6 is selected, we consider all the direct paths from vertex 6. The direct paths from vertex 6 are 6 to 2, 6 to 3, and 6 to 7.

First, we consider the vertex 2. Consider the vertex 6 as 'x', and the vertex 2 as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (11 + 4) < 12

= 15 < 12

Since 15 is not less than 12, we will not update d(2) from 12 to 15

Now we consider the vertex 3. Consider the vertex 6 as 'x', and the vertex 3 as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (11 + 14) < ∞

= 25 < ∞

Since 25 is less than ∞, so we will update d(3) from ∞ to 25.

Now we consider the vertex 7. Consider the vertex 6 as 'x', and the vertex 7 as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (11 + 10) < ∞

= 22 < ∞

Since 22 is less than ∞ so, we will update d(7) from ∞ to 22.

Till now, nodes 0, 1, 4, 5, and 6 have been selected. Now we have to compare all the unvisited nodes, i.e., 2, 3, 7, and 8. Since node 2 has the minimum value, i.e., 12 among all the other unvisited nodes. Therefore, node 2 is selected.

Since node 2 is selected, so we consider all the direct paths from node 2. The direct paths from node 2 are 2 to 8, 2 to 6, and 2 to 3.

First, we consider the vertex 8. Consider the vertex 2 as 'x' and 8 as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (12 + 2) < 15

= 14 < 15

Since 14 is less than 15, we will update d(8) from 15 to 14.

Now, we consider the vertex 6. Consider the vertex 2 as 'x' and 6 as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (12 + 4) < 11

= 16 < 11

Since 16 is not less than 11 so we will not update d(6) from 11 to 16.

Now, we consider the vertex 3. Consider the vertex 2 as 'x' and 3 as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (12 + 7) < 25

= 19 < 25

Since 19 is less than 25, we will update d(3) from 25 to 19.

Till now, nodes 0, 1, 2, 4, 5, and 6 have been selected. We compare all the unvisited nodes, i.e., 3, 7, and 8. Among nodes 3, 7, and 8, node 8 has the minimum value. The nodes which are directly connected to node 8 are 2, 4, and 5. Since all the directly connected nodes are selected so we will not consider any node for the updation.

The unvisited nodes are 3 and 7. Among the nodes 3 and 7, node 3 has the minimum value, i.e., 19. Therefore, the node 3 is selected. The nodes which are directly connected to the node 3 are 2, 6, and 7. Since the nodes 2 and 6 have been selected so we will consider these two nodes.

Now, we consider the vertex 7. Consider the vertex 3 as 'x' and 7 as 'y'.

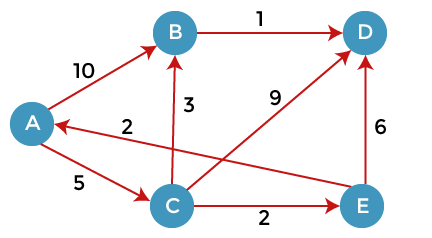
d(x, y) = d(x) + c(x, y) < d(y)

= (19 + 9) < 21

= 28 < 21

Since 28 is not less than 21, so we will not update d(7) from 28 to 21.

Let's consider the directed graph.



Here, we consider A as a source vertex. A vertex is a source vertex so entry is filled with 0 while other vertices filled with ∞. The distance from source vertex to source vertex is 0, and the distance from the source vertex to other vertices is ∞.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | C | D | E |
| ∞ | ∞ | ∞ | ∞ | ∞ |

We will solve this problem using the below table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | 0 | ∞ | ∞ | ∞ | ∞ |

Since 0 is the minimum value in the above table, so we select vertex A and added in the second row shown as below:

As we can observe in the above graph that there are two vertices directly connected to the vertex A, i.e., B and C. The vertex A is not directly connected to the vertex E, i.e., the edge is from E to A. Here we can calculate the two distances, i.e., from A to B and A to C. The same formula will be used as in the previous problem.

If(d(x) + c(x, y)  < d(y))

  Then we update d(y) = d(x) + c(x, y)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
|  |  | 10 | 5 | ∞ | ∞ |

As we can observe in the third row that 5 is the lowest value so vertex C will be added in the third row.

We have calculated the distance of vertices B and C from A. Now we will compare the vertices to find the vertex with the lowest value. Since the vertex C has the minimum value, i.e., 5 so vertex C will be selected.

Since the vertex C is selected, so we consider all the direct paths from the vertex C. The direct paths from the vertex C are C to B, C to D, and C to E.

First, we consider the vertex B. We calculate the distance from C to B. Consider vertex C as 'x' and vertex B as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (5 + 3) < ∞

= 8 < ∞

Since 8 is less than the infinity so we update d(B) from ∞ to 8. Now the new row will be inserted in which value 8 will be added under the B column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
|  |  | 10 | 5 | ∞ | ∞ |
|  |  | 8 |  |  |  |

We consider the vertex D. We calculate the distance from C to D. Consider vertex C as 'x' and vertex D as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (5 + 9) < ∞

= 14 < ∞

Since 14 is less than the infinity so we update d(D) from ∞ to 14. The value 14 will be added under the D column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C |  | 10 | 5 | ∞ | ∞ |
|  |  | 8 |  | 14 |  |

We consider the vertex E. We calculate the distance from C to E. Consider vertex C as 'x' and vertex E as 'y'.

d(x, y) = d(x) + c(x, y) < d(y)

= (5 + 2) < ∞

= 7 < ∞

Since 14 is less than the infinity so we update d(D) from ∞ to 14. The value 14 will be added under the D column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C |  | 10 | 5 | ∞ | ∞ |
|  |  | 8 |  | 14 | 7 |

As we can observe in the above table that 7 is the minimum value among 8, 14, and 7. Therefore, the vertex E is added on the left as shown in the below table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C |  | 10 | 5 | ∞ | ∞ |
| E |  | 8 |  | 14 | 7 |

The vertex E is selected so we consider all the direct paths from the vertex E. The direct paths from the vertex E are E to A and E to D. Since the vertex A is selected, so we will not consider the path from E to A.

Consider the path from E to D.

d(x, y) = d(x) + c(x, y) < d(y)

= (7 + 6) < 14

= 13 < 14

Since 13 is less than the infinity so we update d(D) from ∞ to 13. The value 13 will be added under the D column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C |  | 10 | 5 | ∞ | ∞ |
| E |  | 8 |  | 14 | 7 |
| B |  | 8 |  | 13 |  |

The value 8 is minimum among 8 and 13. Therefore, vertex B is selected. The direct path from B is B to D.

d(x, y) = d(x) + c(x, y) < d(y)

= (8 + 1) < 13

= 9 < 13

Since 9 is less than 13 so we update d(D) from 13 to 9. The value 9 will be added under the D column.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C |  | 10 | 5 | ∞ | ∞ |
| E |  | 8 |  | 14 | 7 |
| B |  | 8 |  | 13 |  |
| D |  |  |  | 9 |  |